CSC258 Week 11

Recap

- Function calls
- Stack, push, pop

Next one

```
int factorial (int n) {
   if (n == 0)
     return 1;
   else
     return n * factorial(n-1);
}
```

Recursion!

Recursion in Assembly

what recursion really is in hardware

I WANNA UNDERSTAND RECURSION
SURE, BUT YOU HAVE TO UNDERSTAND IT FIRST
imgflip.com

```
factorial(3)
  p = 3 * factorial(2)
    factorial(2)
      p = 2 * factorial(1)
        factorial(1)
          p = 1*factorial(0)
            factorial (0)
              p = 1 # Base!
              return p

          return p

      return p

  return p
```

```
int factorial(int n) {
    if (n==0)
      return 1;
    else
      return n*factorial(n-1);
}
```

Before writing assembly, we need to know explicitly where to store values

```
int factorial (int n) {
   if (n == 0)
      return 1;
   else
      return n * factorial(n-1);
}
```

Need to store …

- the value of n
- the value of n – 1
- the value factorial(n-1)
- the return value: 1 or n*factorial(n-1)

Design decision #1: store values in registers

```
int factorial(int n) {
   if (n==0)
     return 1;
   else
     return n*fact(n-1);
}
```

Does it work?

- store n in $to
- store n-1 in $t1
- store factorial(n-1) in $t2
- store return value in $t3

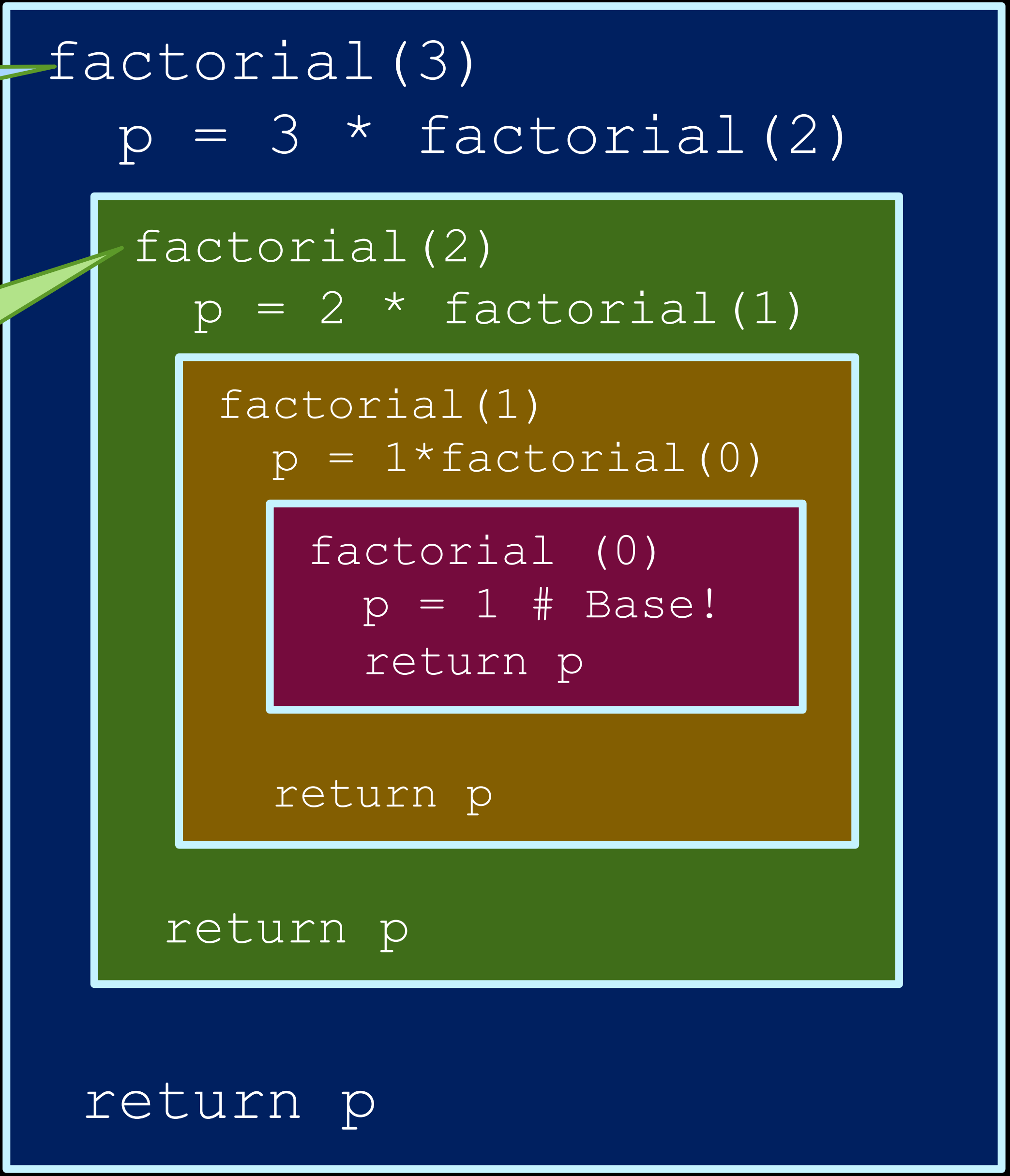
No, it doesn't work.

Store **n=3** in $t0

Store **n=2** in $t0, the stored 3 is overwritten, lost!

Same problem for $t1, t2, t3

- store n in $t0
- store n-1 in $t1
- store factorial(n-1) in $t2
- store return value in $t3



```
factorial(3)
  p = 3 * factorial(2)
    factorial(2)
      p = 2 * factorial(1)
        factorial(1)
          p = 1*factorial(0)
            factorial (0)
              p = 1 # Base!
              return p

          return p

      return p

  return p
```

A register is like a laundry basket -- you put your stuff there, but when you call another function (person), that person will use the same basket and take / mess up your stuff.

And yes, the other person will guarantee to use the same basket because … the other person is **YOU**! (because recursion)

So the correct design decision is to use __**Stack**__ .

Each recursive call has its own space for storing the values

Stores **n=2** for factorial (2)

Stores **n=3** for factorial (3)

Stacking Laundry Baskets are ideal for sorting laundry in small spaces.

Two useful things about stack

1. It has a lot of space
2. Its LIFO order (last in first out) is suitable for implementing recursions (function calls).

LIFO order & recursive calls

Note: Everybody is getting the **correct** basket because of LIFO!

```
factorial(2)
  p = 2 * factorial(1)
    factorial(1)
      p = 1*factorial(0)
        factorial (0)
          p = 1 # Base!
          return p

      return p

  return p
```

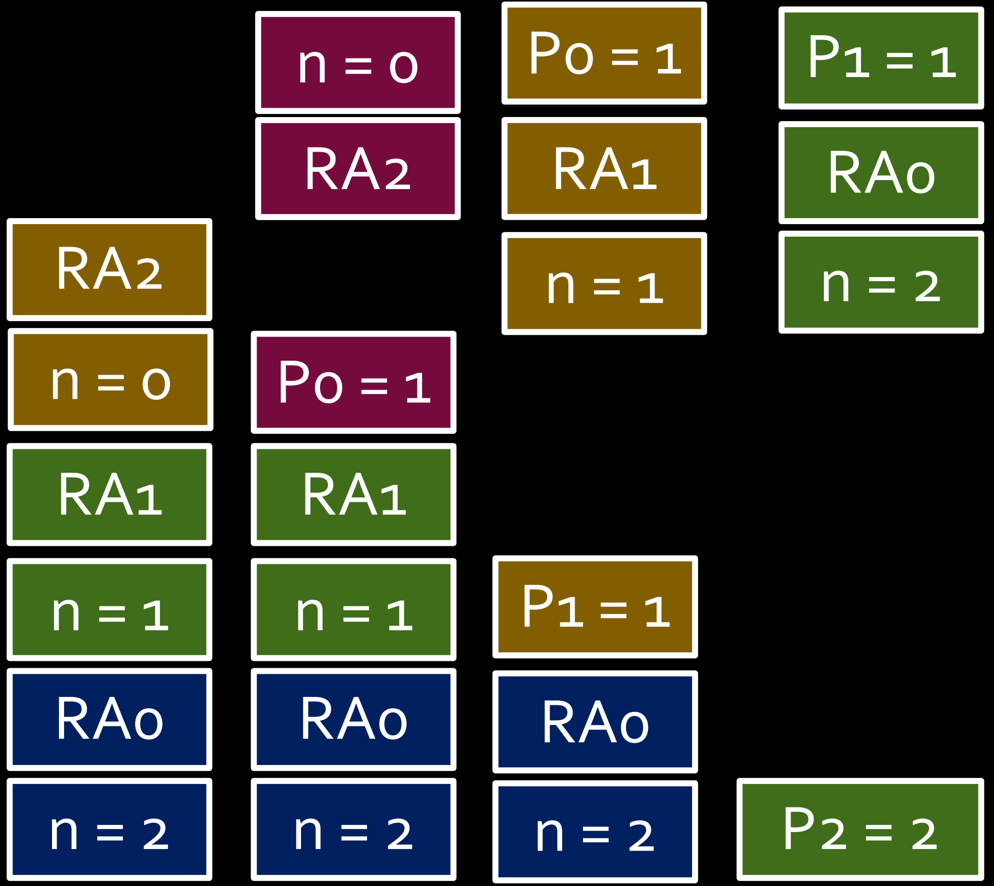
n = 0
n = 1
n = 2
Stacking Laundry Baskets are ideal for sorting laundry in small spaces.

Design decisions made,
now let’s actually write the
assembly code

LIFO order & recursive calls

```
int x = 2;
int y = factorial(x)
print(y) # RA0
```

```
factorial(n=2)
  r = factorial(1)
    factorial(n=1)
      r = factorial(0)
        factorial(n=0)
          p = 1 # Base!
          return p #P0
      p = n * r;  # RA2
    return p #P1
  p = n * r; # RA1
return p # P2
```



Actions in factorial (n)

Before making the recursive call

- pop argument n
- push argument n-1 (arg for recursive call)
- push return address (remember where to return)
- make the recursive call

After finishing the recursive call

- pop return value from recursive call
- pop return address
- compute return value
- push return value (so the upper call can get it)
- jump to return address

factorial(int n)

n → $t0
n-1 → $t1
fact(n-1) → $t2

- Pop n off the stack
  - Store in $t0
- If $t0 == 0,
  - Push return value 1 onto stack
  - Return to calling program
- If $t0 != 0,
  - Push $t0 and $ra onto stack
  - Calculate n-1
  - Push n-1 onto stack
  - Call factorial
    - …time passes…
  - Pop the result of factorial (n-1) from stack, store in $t2
  - Restore $ra and $t0 from stack
  - Multiply factorial (n-1) and n
  - Push result onto stack
  - Return to calling program

factorial(int n)

n → $t0

n-1 → $t1

fact(n-1) → $t2

```
fact:         lw $t0, 0($sp)
              addi $sp, $sp, 4
              bne $t0, $zero, not_base
              addi $t0, $zero, 1
              addi $sp, $sp, -4
              sw $t0, 0($sp)
              jr $ra
not_base:     addi $sp, $sp, -4
              sw $t0, 0($sp)
              addi $sp, $sp, -4
              sw $ra, 0($sp)
              addi $t1, $t0, -1
              addi $sp, $sp, -4
              sw $t1, 0($sp)
              jal fact
```

- Pop n off the stack
  - Store in $t0
- If $t0 == 0,
  - Push return value 1 onto stack
  - Return to calling program
- If $t0 != 0,
  - Push $t0 and $ra onto stack
  - Calculate n-1
  - Push n-1 onto stack
  - Call factorial
  - Pop the result of factorial (n-1) from stack, store in $t2
  - Restore $ra and $t0 from stack
  - Multiply factorial (n-1) and n
  - Push result onto stack
  - Return to calling program

Note: codes on the slides are not guaranteed to be correct. You need to be able to find the errors and fix them.

factorial(int n)

n → $to

n-1 → $t1

fact(n-1) → $t2

```
lw $t2, 0($sp)
addi $sp, $sp, 4
lw $ra, 0($sp)
addi $sp, $sp, 4
lw $t0, 0($sp)
addi $sp, $sp, 4
mult $t0, $t2
mflo $t3
addi $sp, $sp, -4
sw $t3, 0($sp)
jr $ra
```

- Pop n off the stack
  - Store in $to
- If $to == 0,
  - Push return value 1 onto stack
  - Return to calling program
- If $to != 0,
  - Push $to and $ra onto stack
  - Calculate n-1
  - Push n-1 onto stack
  - Call factorial
  - Pop the result of factorial (n-1) from stack, store in $t2
  - Restore $ra and $to from stack
  - Multiply factorial (n-1) and n
  - Push result onto stack
  - Return to calling program

Note: codes on the slides are not guaranteed to be correct. You need to be able to find the errors and fix them.

Recursive programs

- Use of stack
  - Before recursive call, store the register values that you use onto the stack, and restore them when you come back to that point.
  - Store $ra as one of those values, to remember where each recursive call should return.

```
int factorial (int x) {
    if (x==0)
        return 1;
    else
        return x*factorial(x-1);
}
```

Translated recursive program (part 1)

```
main:           addi    $t0, $zero, 10        # call fact(10)
                addi    $sp, $sp, -4          #   by putting 10
                sw      $t0, 0($sp)           #   onto stack
                jal     factorial             # result will be
                ...                           #   on the stack
```

```
factorial:      lw      $a0, 4($sp)           # get x from stack
                bne     $a0, $zero, rec       # base case?
base:           addi    $t0, $zero, 1         # put return value
                sw      $t0, 4($sp)           #   onto stack
                jr      $ra                   # return to caller
rec:            addi    $sp, $sp, -4          # store return
                sw      $ra, 0($sp)           #   addr on stack
                addi    $a0, $a0, -1          # x--
                addi    $sp, $sp, -4          # push x on stack
                sw      $a0, 4($sp)           #   for rec call
                jal     factorial             # recursive call
```

Note: codes on the slides are not guaranteed to be correct. You need to be able to find the errors and fix them.

Translated recursive program (part 2)

```
(continued from part 1)
                lw     $v0, 0($sp)          # get return value
                addi   $sp, $sp, 4          #   from stack
                lw     $ra, 0($sp)          # restore return
                addi   $sp, $sp, 4          #   address value
                lw     $a0, 0($sp)          # restore x value
                addi   $sp, $sp, 4          #   for this call
                mult   $a0, $v0             # x*fact(x-1)
                mflo   $t0                  # fetch product
                addi   $sp, $sp, -4         # push product
                sw     $t0, 0($sp)          #   onto stack
                jr     $ra                  # return to caller
```

Note: codes on the slides are not guaranteed to be correct. You need to be able to find the errors and fix them.

- Note: `jal` always stores the next address location into `$ra`, and `jr` returns to that address.

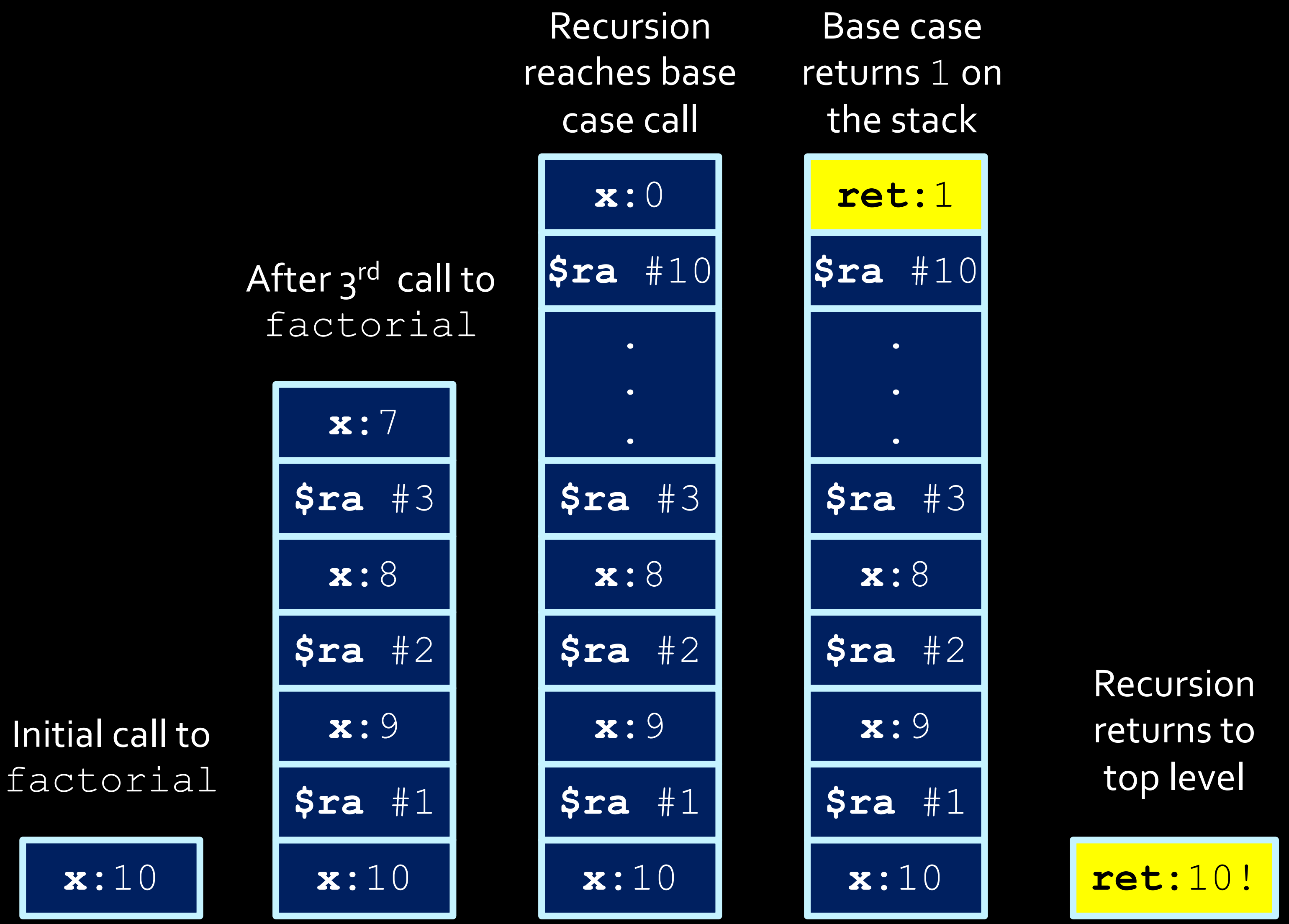
Assembly doesn’t support recursion

- Assembly programs are just a linear sequence of assembly instructions, where you jump to the beginning of the program over and over again…

Recursion comes from the stack

- …while sensibly storing and retrieving remembered values from the stack

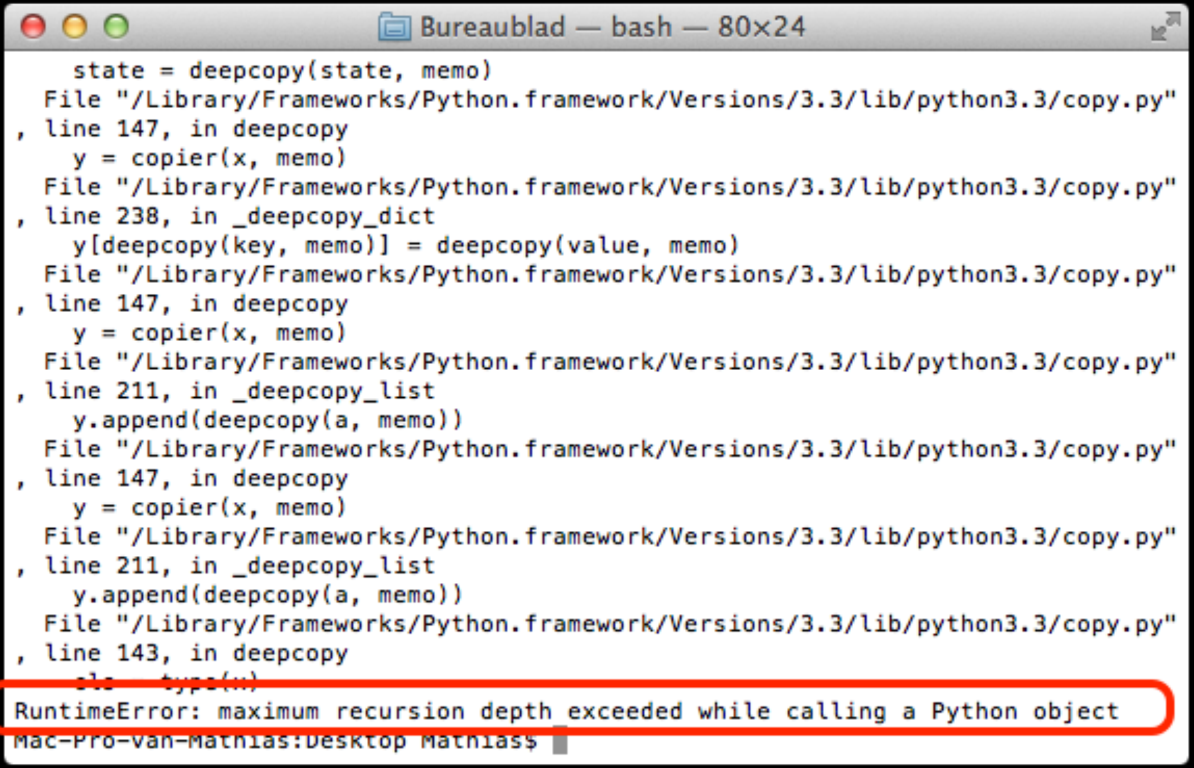
Factorial stack view


Initial call to factorial
x:10
After 3rd call to factorial
x:7
$ra #3
x:8
$ra #2
x:9
$ra #1
x:10
Recursion reaches base case call
x:0
$ra #10
. . .
$ra #3
x:8
$ra #2
x:9
$ra #1
x:10
Base case returns 1 on the stack
ret:1
$ra #10
. . .
$ra #3
x:8
$ra #2
x:9
$ra #1
x:10
Recursion returns to top level
ret:10!

You *can* recurse too much

The stack is NOT of infinite size, so there is always a limit on the number of recursive calls that you can make.

When exceeds that limit, you get a stack overflow, all content of the stack will be dumped.



```
    state = deepcopy(state, memo)
  File "/Library/Frameworks/Python.framework/Versions/3.3/lib/python3.3/copy.py"
, line 147, in deepcopy
    y = copier(x, memo)
  File "/Library/Frameworks/Python.framework/Versions/3.3/lib/python3.3/copy.py"
, line 238, in _deepcopy_dict
    y[deepcopy(key, memo)] = deepcopy(value, memo)
  File "/Library/Frameworks/Python.framework/Versions/3.3/lib/python3.3/copy.py"
, line 147, in deepcopy
    y = copier(x, memo)
  File "/Library/Frameworks/Python.framework/Versions/3.3/lib/python3.3/copy.py"
, line 211, in _deepcopy_list
    y.append(deepcopy(a, memo))
  File "/Library/Frameworks/Python.framework/Versions/3.3/lib/python3.3/copy.py"
, line 147, in deepcopy
    y = copier(x, memo)
  File "/Library/Frameworks/Python.framework/Versions/3.3/lib/python3.3/copy.py"
, line 211, in _deepcopy_list
    y.append(deepcopy(a, memo))
  File "/Library/Frameworks/Python.framework/Versions/3.3/lib/python3.3/copy.py"
, line 143, in deepcopy
    cls = type(x)
RuntimeError: maximum recursion depth exceeded while calling a Python object
Mac-Pro-van-Mathias:Desktop Mathias$
```

Supporting Recursion in General

- The process we've defined is ad hoc
- We stored an argument on the stack. We saved the RA register.

But how do you support recursion generally?

- You must know the signature of the function you're calling. The number of arguments is key so that you how many things to pop from the stack.
  - This is why C has function prototypes.
- You need to store the values of all of the registers that you use.

Optimization: Caller and Callee Saves

- To reduce the number of registers that need to be saved, MIPS uses caller save and callee save registers.

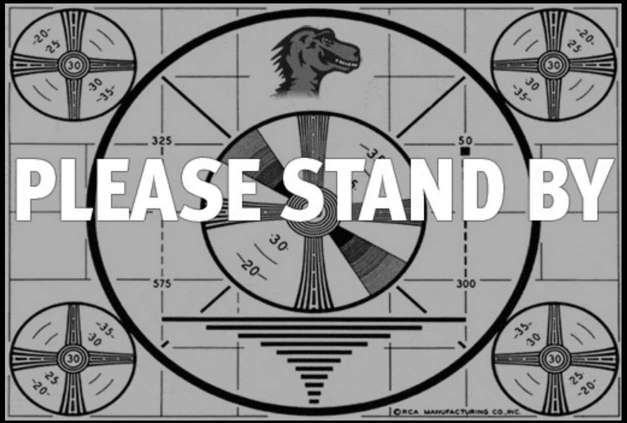
- The **t** registers are caller save: if you are using them and want to keep the value, save it before calling the function.
- The s registers are callee save: if you want to use them, you should save the values before using them.

What advantage does this scheme have?

Interrupts and Exception

A note on interrupts

- Interrupts take place when an external event requires a change in execution.
  - Example: arithmetic overflow, system calls (`syscall`), Ctrl-C, undefined instructions.
  - Usually signaled by an external input wire, which is checked at the end of each instruction.
  - High priority, override other actions



A note on interrupts

- Interrupts can be handled in two general ways:
  - Polled handling: The processor branches to the address of interrupt handling code (interruption handler), which begins a sequence of instructions that check the cause of the exception, i.e., need to ask around to figure out what type of exception.

    → This is what MIPS uses (syscall → CPU checks `v0, etc`)
  - Vectored handling: The processor can branch to a different address for each type of exception. Each exception address is separated by only one word. A jump instruction is placed at each of these addresses for the handler code for that exception. So no need to ask around.

Interrupt Handling

- In the case of polled interrupt handling, the processor jumps to exception handler code, based on the value in the cause register (see table).
  - If the original program can resume afterwards, this interrupt handler returns to program by calling **`rfe`** instruction.
  - Otherwise, the stack contents are dumped and execution will continue elsewhere.
  - The above happens in kernel mode.

0 (INT)	external interrupt.
4 (ADDRL)	address error exception (load or fetch)
5 (ADDRS)	address error exception (store).
6 (IBUS)	bus error on instruction fetch.
7 (DBUS)	bus error on data fetch
8 (Syscall)	Syscall exception
9 (BKPT)	Breakpoint exception
10 (RI)	Reserved Instruction exception
12 (OVF)	Arithmetic overflow exception

Interrupt Handling

- The exception handler is just assembly code.
  - just like any other function
  - … but it must NOT cause an error! (There is no one to handle it)

- One particularly useful error handler
  - In the old days, error handling code useful take up 80% of the OS code.
  - Many error handlings were later unified into one way
  - General solution: "kernel panic" -- dump information and ask human to reboot the computer.

Parallelism

Parallelism

- Parallelism is the idea that you can derive benefit from completing multiple tasks simultaneously.

Performance

When we discuss performance, we often consider the following two metrics:

- **Latency**: the length of time required to perform an operation.
  - How long it takes to travel from A to B on Highway 401
  - More about a single task. We learned about the timing analysis.
- **Throughput**: the number of operations that can be completed within a unit of time.
  - How many cars arrive at B from A via Highway 401 per hour
  - More about multiple tasks.
  - Think about how your computer's graphics card work. It tries to process many pixels simultaneously.

Types of Parallelism in Hardware

- **Spatial**: Completing the same task multiple times at the same time.
- **Temporal (pipelined)**: Breaking a task into pieces, so that multiple different instructions can be in process at the same time.

Don’t confuse this with locality!

Spatial Parallelism

Temporal Parallelism

Spatial vs Temporal Parallelism (pic from DDCA)

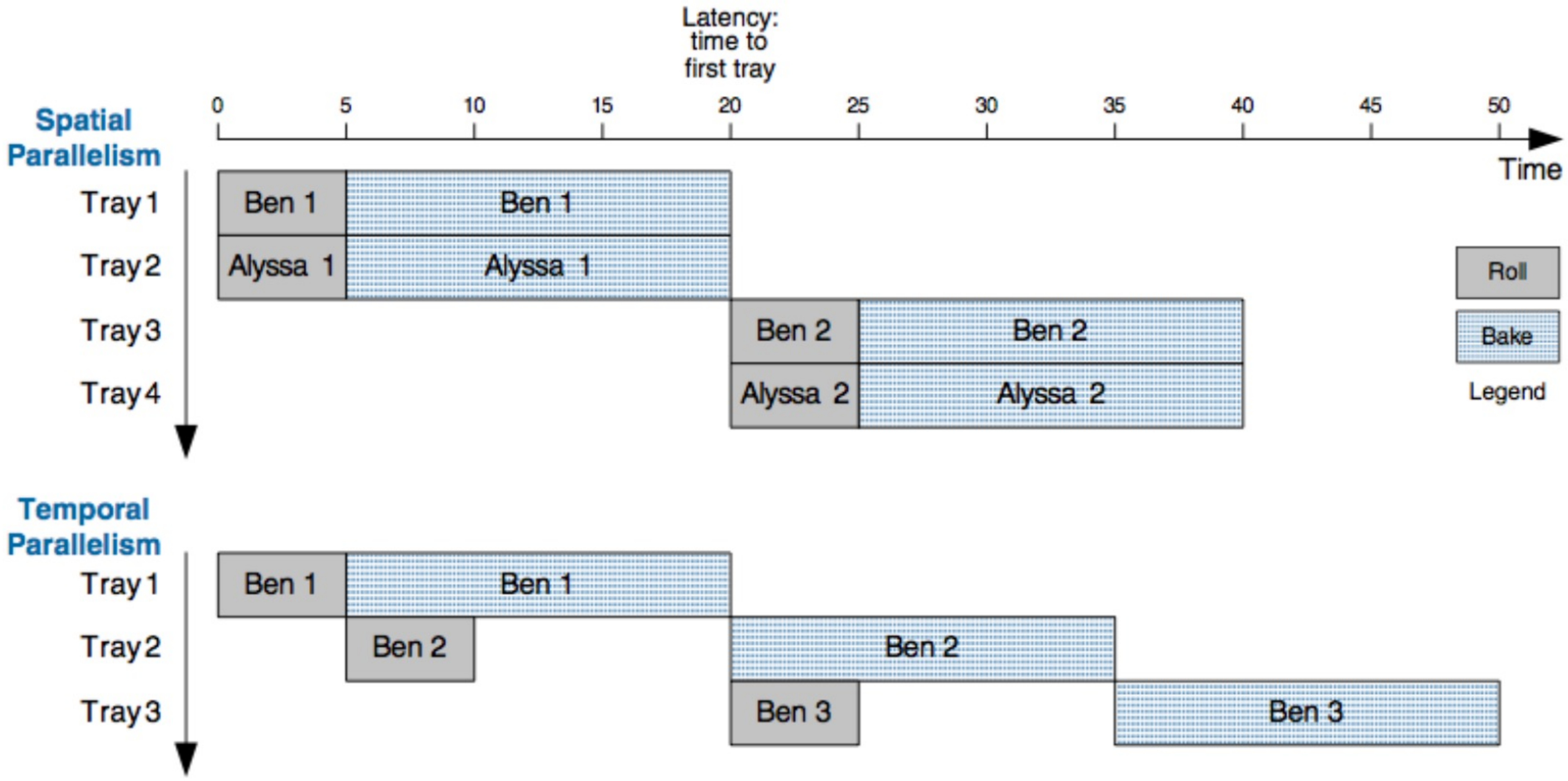


Figure 3.55 Spatial and temporal parallelism in the cookie kitchen

Pipelined Microarchitectures

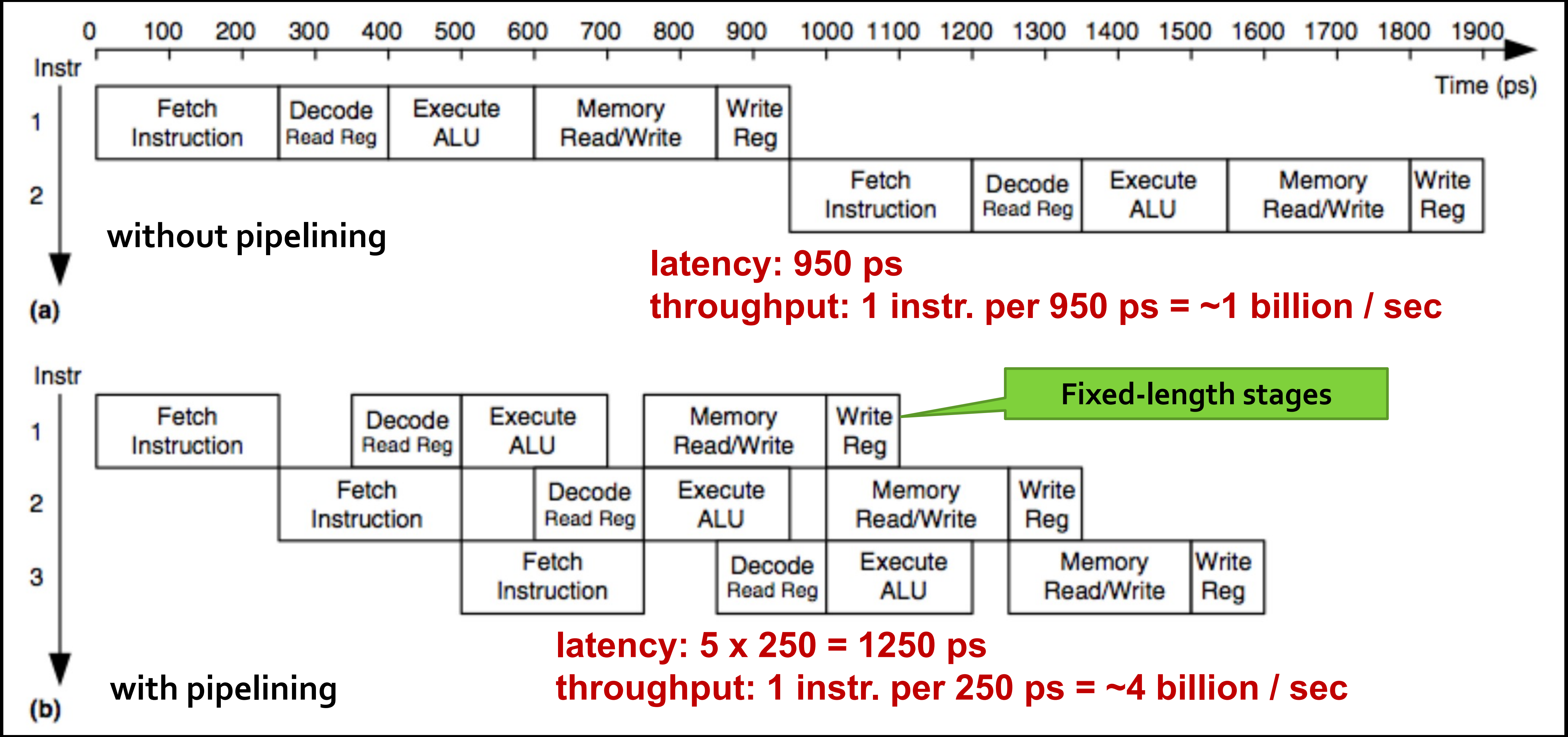
Review: Executing a Program

- First, load the program into memory.
- Set the program counter (PC) to the first instruction in memory and set the SP to the first empty space on the stack
- Let instruction fetch/decode do the work! The processor can control what instruction is executed next.
- When the process needs support from the operating system (OS), it will “trap” (“throw an exception”)

Execution Stages

- **Fetch**: Updating the PC and locating the instruction to execute.
- **Decode**: Translating the instruction and reading inputs from the register file.
- **Execute** / Address Computation: Using the ALU to compute an operation or calculate an address.
- **Memory Read or Write**: Memory operations must access memory. Non-memory operations skip this.
- **Register Writeback**: The result is written to the register file.

Pipelining the Execution Stages



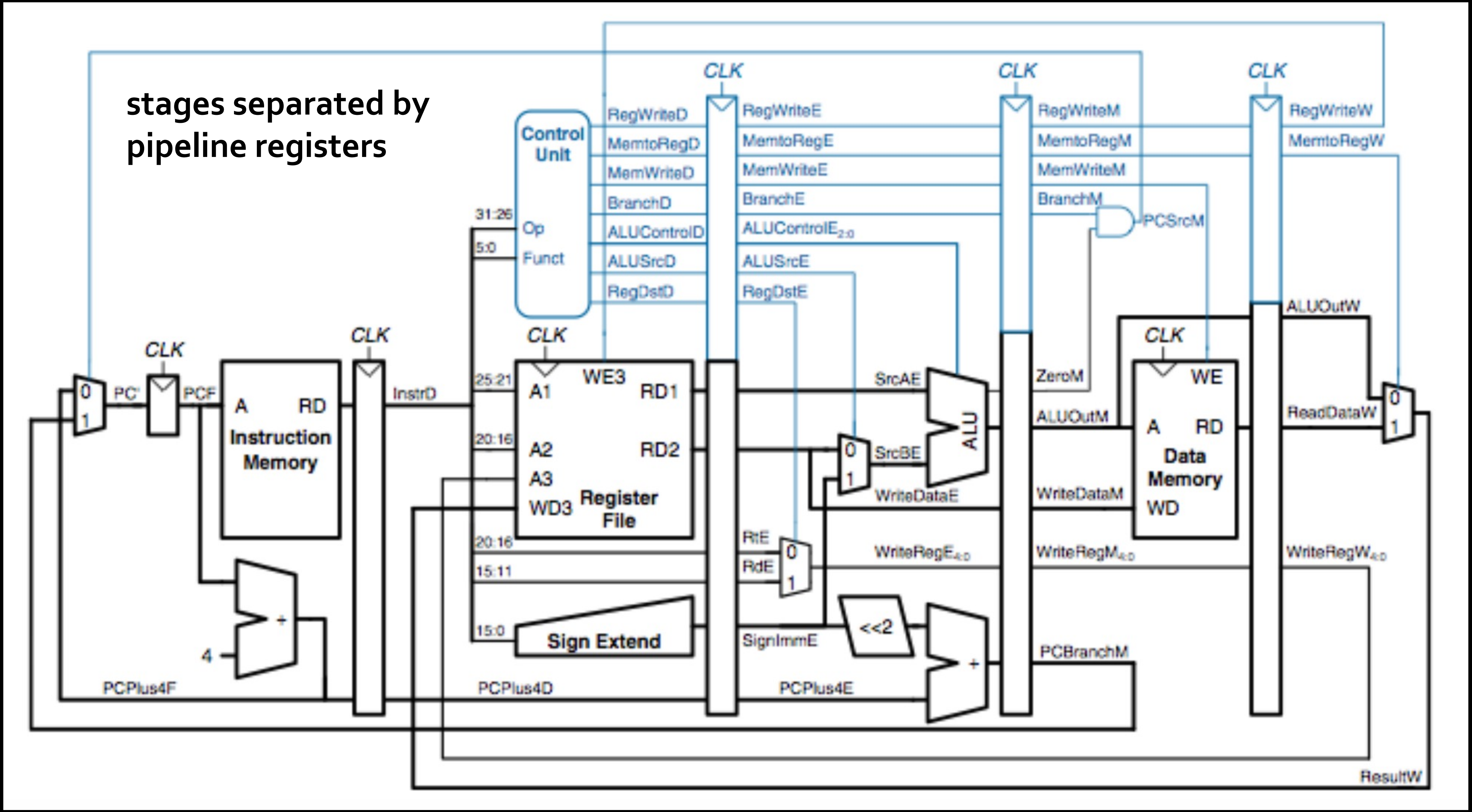
without pipelining

latency: 950 ps
throughput: 1 instr. per 950 ps = ~1 billion / sec

with pipelining

latency: 5 x 250 = 1250 ps
throughput: 1 instr. per 250 ps = ~4 billion / sec

Pipelined Datapath



Hazard

- What happens if an instruction needs a value that has not been computed?
  - This is a **data hazard**.
  - Example: `$t0 += 2 followed by $t0 += 3`
- What if an instruction is changing the PC? Shouldn't it complete before we fetch another instruction?
  - This is a **control hazard**
  - can happen when branching or jumping.

Mitigating Hazards

- **Data forwarding**: a. k. a bypassing, values are available before they are written back, i.e., after the execute stage, results are available, and they can be forwarded to the stage that needs them.
  - Don't wait until MEM READ/WRITE or WRITE REG to finish!
  - Requires some additional wiring in the CPU
- **Stalls**: Sometimes, you just have to wait.
  - A stall (or no-op) keeps a pipeline stage from doing anything.

Stalls and Performance

- Stalls throttle performance.

- Sometimes, we can predict a result.
  - e.g., branch prediction
  - If we're correct, then we get a performance win.
  - If we're wrong, we "drop" the instruction that is using predicted values, and we're *almost* no worse off.
  - Prediction is big business. It consumes a huge amount of the chip.

Summary: Pipelining

- The pipelined design traded space for time: it added additional hardware to increase throughput.